Automata and Finite Automata

By C. Y. LEE

(Manuscript received March 17, 1960)

Since it is not clear, in general, how an automaton should best be characterized, one of the purposes of this paper is to find ways to go from one characterization to another. In doing so, we have not been completely impartial—the programming approach has been emphasized more than the others. There are perhaps two reasons for this emphasis: First and the more obvious one is the closeness between theoretical programming discussed here and programming of digital computers. Secondly, the programming approach has provided a way of looking at automata that seems to make certain ideas less obscure—the construction of a universal program in Section III of this paper is one such example. In the theory of finite automata, Theorem 3 is an attempt to unify the ideas of complete and partial automata, which have generally been treated separately in the past.

1. INTRODUCTION

The invention of modern computers seems to have been anticipated by many years by Turing.[1] Yet it is remarkable how little the progress of computers has been influenced by Turing's work. There is, perhaps, a basic difference in viewpoint that may account for this lack of convergence. Turing looked at machines from the point of view of their internal behavior. Although Turing originated the concept of universal machines, his idea seems to correspond much closer to that of our special-purpose machines. Every machine, by virtue of its state description, performs a specific task; a machine is altered only if its internal structure is altered. Computers, on the other hand, are generally specified in terms of their external capabilities. Their internal structure remains more or less fixed once they come into being. A computer is then a universal machine in disguise, and every Turing machine corresponds to a particular computer program. One may therefore study the behavior and structure of programs rather than work with states.

The first step in this direction was perhaps taken by Wang,[2] who based his ideas of machines on a computer (which he called a B-machine) that

had four kinds of instructions: move to the right or left; mark; transfer conditionally. A B-machine is close to the ultimate in simplicity, but is still capable of computing everything that a Turing machine is capable of and, with a suitable program, is capable of being universal.

As a model, B-machines are attractive because of their intrinsic simplicity. On the other hand, because a B-machine does not have the ability to erase, it is very difficult to write even fairly simple programs without having to work out intricate details. In this paper we have, therefore, introduced a modified B-machine—one which is empowered with the ability to erase. We have called a machine of this kind a W-machine.

The similarities and differences between W-machines and two-symbol Turing machines are shown in Sections II and III. In Section IV we describe the construction of a universal W-machine to show the kinds of techniques involved in W-machine programming. It may be interesting to note here that, once a few useful subprograms are written, the main linkage program takes but a few instructions. Because of its simplicity, one may suspect that it is harder to construct sophisticated combinatorial or symbol-operation kinds of programs on a W-machine than it is on a more complex computer. But we would not be surprised if such a suspicion turns out to be groundless; what makes a W-machine a poor computer may well be only its disregard for time.

The subfamily of W-machines in which each machine has a bounded memory constitutes the family of finite automata. Because finite automata are abstract models of sequential switching circuits, there bas been much current interest in their behavior. As a result, there have been a number of approaches to problems in connection with finite automata. In Section V it is shown that finite automata may be characterized by the deletion of one of the five kinds of W-machine instructions. There is thus a program analog of finite automata.

In Section VI the relation between finite automata and sets of input sequences is discussed. Among other things we present within our framework a result of Kleene[3] that makes it possible to represent finite automata by algebraic-like expressions. This characterization seems very natural in many ways, except that the expressions can easily get very lengthy. The problem of how best to handle these expressions appears very intriguing and, as far as we know, is quite open.

II. TUNING MACHINES

A machine will be called an A-machine if it consists, aside from its control mechanism, of the following:

i. A one-way potentially infinite tape (say infinite to the right) divided into squares. Each square can either be marked (having in it the symbol 1) or erased (having in it the symbol 0), and

ii. A reading and writing head that scans some square of the tape at any discrete moment of time. Since the tape is finite to the left, the machine is assumed to stop if the read-write head is ordered to go to the left of the leftmost square of the tape.

The content c_0 of the tape of the A-machine previous to the initial moment of time, consisting of a finite sequence of zeros and ones, is called the (tape) *input* to the A-machine. As time advances, the tape content would change unless some stable condition is reached, so that we would get a sequence c of tape contents $(c_0, c_1, \cdots)$, where c_j is a later tape content than c_i if $i < j$, and where $c_i \neq c_{i+1}$. The sequence c is called the *external behavior* of the A-machine relative to the tape input c_0. Two A-machines are said to be *completely equivalent* if they have identical external behaviors relative to all tape inputs. That is, two A-machines are completely equivalent if they cannot be distinguished by anyone observing just the sequence of tape contents.

The idea of complete equivalence is too stringent at times. If an A-machine is used to compute values of a function, what the machine does while it is processing its data is, in a sense, irrelevant as long as the final answer turns out to be the desired answer. We will, later on, also consider a less stringent type of equivalence.

The fact that an A-machine has a potentially infinite tape implies that it has an indefinitely large memory. It might be helpful to keep the notion that the tape is finite at any moment, but that at any moment a finite amount of blank tape may be added to the right whenever such a demand arises. In the same way, it is helpful to note that every input is a finite sequence of zeros and ones. We will, however, speak of the *null input*, meaning a string of zeros indefinitely long. The null input corresponds to an indefinitely long blank tape.

We will consider the following model of a Turing machine, hereafter called a T-machine, as one of the A-machines. In addition to being an A-machine, it has k active internal states $q_1, q_2, \cdots, q_k$ and an inactive state q_0 in which the machine is assumed to stop. The machine can have one of the following combination of actions: erase or mark the square under scan; move the read-write head one square to the left or one square to the right; go into some state q_j. A T-machine is completely specified if its combination of actions is specified for every state of the machine and each of the two symbols under scan, and if the initial state and the initial square under scan are given.

For instance, the following one-state (i.e. one active state) T-machine, if started initially scanning a square in the interior of its tape, will have its read-write head swinging back and forth, changing ones to zeros while going in one direction and changing zeros to ones while going in the other direction. The read-write head will either proceed indefinitely to the right or will eventually stop at the leftmost square. In this and later description of A-machines, we will use the letter m to denote the action of marking the square under scan; e for the action of erasing the square under scan; $+$ for the action of moving the read-write head one square to the right of the square under scan; and $-$ for the action of moving the read-write head one square to the left:

State	Symbol	
	0	1
$*q$	$m, +, q$	$e, -, q$

Here q designates the single active state of the T-machine, and * denotes the fact that q is also the initial state of this machine. If the square under scan is not marked, a mark is put in it, the read-write head moves one square to the right, and the machine returns to state q. If the square under scan is marked, it is then erased, the read-write head moves one square to the left, and the machine again returns to state q.

From now on, we will at times use the notation q_i ; m or e, $+$ or $-$, q_j ; m or e, $+$ or $-$, q_k for each combination of actions of any T-machine. Thus, the combination of actions of the one-state T-machine in question can be written: q; m, $+$, q; e, $-$, q.

III. W-MACHINES

A W-machine is an A-machine together with a program made up of an ordered list of the following five types of base instructions: (a) e: erase the square under scan; (b) m: mark the square under scan; (c) $+$: move the read-write head one square to the right; (d) $-$: move the read-write head one square to the left; and (e) $t(A)$: transfer to program address A if the square under scan is marked, otherwise transfer to the next program address on the ordered list. These base instructions are executed in order by a control mechanism. The initial program address and the initial square under scan are given.

A program of a W-machine consisting of all base instructions with each instruction having a separate address is called a *base program*. Let us consider a W-machine completely equivalent to the one-state T-

machine illustrated earlier. The base program for this machine is

1. $t7$
2. m
3. $+$
4. $t7$
5. m
6. $t2$
7. e
8. $-$
9. $t7$
10. m
11. $t2$.

We note that the instructions in the program refer to only two addresses, address 2 and address 7. The program may therefore be equally well written

1. $t3$
2. $m, +, t3, m, t2$
3. $e, -, t3, m, t2,$

where the instructions contained in one line are understood to be executed consecutively. This notation simplifies the writing of W-machine programs and will be used in this paper wherever it is convenient to do so.

A base program of a W-machine is said to be *minimal* if there is no W-machine completely equivalent to it with fewer base instructions in its program. In order not to have to consider special cases later, let us agree at this stage to rule out certain trivial redundancies in W-machine programs. Consider two W-machines, W_1 and W_2, as follows:

Machine W_1
1. m
2. $+$
3. $t1$

Machine W_2
1. m
2. e
3. m
4. $+$
5. $t1$.

Machines W_1 and W_2 are not completely equivalent, since they have nonidentical external behavior. The difference is, however, of a minor nature. We will therefore agree that, whenever a W-machine program contains consecutive instructions

A. e or m
$A + 1$. e or m
$\vdots$
$A + i$. e or m,

only the last instruction (in address $A + i$) will be retained, and the others will be deleted. Furthermore, if it should become necessary to mark and erase a square in succession, the final symbol in that square will be accepted as the output symbol for that moment.

The fact that a base program is minimal itself implies that the base program cannot contain certain subprograms.

Lemma 1:

Let P be a minimal base program of a W-machine. Then P cannot have two consecutive addresses A and $A + 1$ having in them the following base instructions:

(i)	$A.$	$t(B)$	(iii)	$A.$	e
	$A + 1.$	$t(C);$		$A + 1.$	$e;$
(ii)	$A.$	e	(iv)	$A.$	m
	$A + 1.$	$t\,(C);$		$A + 1.$	$m.$

Proof: In (i) and (ii), if address $A + 1$ is never referred to, P cannot be minimal since the $(A + 1)$th instruction can be deleted. On the other hand, if there is some instruction $t(A + 1)$ in P, such an instruction can be changed to $t(C)$, again making the $(A + 1)$th instruction superfluous. This proves (i) and (ii); (iii) and (iv) are obvious, and the lemma follows.

Theorem 1:

I. Given a W-machine having b base instructions, there is a completely equivalent T-machine with not more than b states.

II. Given a T-machine with s states, there is a completely equivalent W-machine with not more than $10s + 1$ base instructions.

Proof: Let a W-machine with b base instructions be given. That there is a completely equivalent T-machine is clear. It remains for us to show for part I of the theorem that b states would suffice.

Let P be a minimal base program for the W-machine and A be the initial address of P. Then, by Lemma 1, the base instructions in addresses A and $A + 1$ are one of the following:

(i)	$A.$	$t(B)$	(iv)	$A.$	m or e
	$A + 1.$	$e;$		$A + 1.$	$+$ or $-;$
(ii)	$A.$	$t(B)$	(v)	$A.$	$+$ or $-$
	$A + 1.$	$m;$		$A + 1.$	m or $e;$
(iii)	$A.$	$t(B)$	(vi)	$A.$	$+$ or $-$
	$A + 1.$	$+$ or $-;$		$A + 1.$	$+$ or $-.$

In (i), (ii) and (iii), we assert that the base instruction in address B can be made one of the following:

$$\text{(a) } B.\quad e \qquad \text{or} \qquad \text{(b) } B.\quad + \text{ or } -.$$

This is true because, if the instruction in B should be $t(C)$ and the instruction in C should be $t(D)$ and so on, then at some point in the chain, say address E, the instruction must be a nontransfer instruction, for otherwise the program would not have been minimal. We may then replace the instruction $t(B)$ in A by $t(E)$. On the other hand, if the instruction in B should be m, then the instruction in A could have been replaced by:

$$A.\quad t(B+1);$$

and the assertion follows.

In (i) and case (a), by Lemma 1, the base instructions in addresses $A+2$ and $B+1$ must be $+$ or $-$. Thus, address A can be associated with a T-machine state

$$q(A);\quad e, + \text{ or } -, q(A+3);\quad e, + \text{ or } -, q(B+2).$$

Similarly, in case (b) address A can be associated with a T-machine state

$$q(A);\quad e, + \text{ or } -, q(A+3);\quad m, + \text{ or } -, q(B+1).$$

It should be noted that a T-machine state may replace more than just address A. For example, in (i) case (a) the T-machine state replaces the five addresses A, $A+1$, $A+2$, B and $B+1$ if none of these addresses is referred to elsewhere in the program. Therefore, in going from a W-machine to a T-machine as described by the procedure outlined here, the T-machine will in general have fewer than b states.

In (ii), the $(A+2)$th instruction can be either

$$A+2.\quad + \text{ or } - \qquad \text{or} \qquad A+2.\quad t(C).$$

The former is no different from (i). In tbe latter, the instruction in address C can be made one of the following:

$$C.\quad e \qquad \text{or} \qquad C.\quad + \text{ or } -.$$

The T-machine states to be associated with address A in case (a) corresponding to these two subcases are respectively

$$q(A);\quad e, + \text{ or } -, q(C+2);\quad e, + \text{ or } -, q(B+2),$$

and

$$q(A);\quad m, + \text{ or } -, q(C+1);\quad e, + \text{ or } -, q(B+2);$$

and in case (b) are respectively

$$q(A); \quad e, + \text{ or } -, q(C + 2); \quad m, + \text{ or } -, q(B + 1),$$

and

$$q(A); \quad m, + \text{ or } -, q(C + 1); \quad m, + \text{ or } -, q(B + 1).$$

Case (iii) is similar to (i). In (iv) and (v), the two addresses, A and $A + 1$, can obviously be associated with a single T-machine state. In (vi), each address A or $A + 1$ may be associated with a single T-machine state. Therefore, there is a completely equivalent T-machine with not more than b states and part I of the theorem follows.

To prove part II, let a T-machine with s states be given with states q_i, $i = 1, 2, \cdots, s$:

$$q_i; \quad a_i(0), b_i(0), q_i(0); \quad a_i(1), b_i(1), q_i(1),$$

where a_i is either m or e and b_i is either $+$ or $-$. Associate with each state q_i two addresses A_i and A_i' of a W-machine:

$$A_i. \quad a_i(0), b_i(0), t[A_i'(0)], m, t[A_i(0)];$$

$$A_i'. \quad a_i(1), b_i(1), t[A_i'(1)], m, t[A_i(1)].$$

Next, if q_j is the initial state of the T-machine, we will add an initial address $A_j - 1$ where we have

$$A_j - 1. \qquad t(A_j').$$

The W-machine so defined is completely equivalent to the T-machine, having exactly $10s + 1$ base instructions. This proves part II of the theorem.

The bound $10s + 1$ on the number of base instructions cannot be lowered if the first address is to be always the initial address of a W-machine program. If we are allowed to begin a program at some intermediate address, the bound $10s + 1$ can be lowered to perhaps $8s + 1$.

From this result, it follows that whatever is true about T-machines is functionally true about W-machines, and conversely. The choice of whether to use the T-machine or the W-machine model is therefore somewhat arbitrary. We have found that the T-machine model is convenient for state description of finite automata (Section V) and the W-machine model more satisfactory for problems involving operations with symbols. The latter contention is illustrated by a universal W-machine described below.

IV. A UNIVERSAL W-MACHINE.

A very interesting result of Turing[1] was his construction of a machine which is capable of imitating any target machine when given a description of the target machine. Such a machine is known as a universal Turing machine. To our knowledge there have been two independent and very ingenious constructions of universal machines which greatly simplified Turing's work. The earlier one is due to Moore[4] and the other to Ikeno.[5] Moore's machine has two symbols and three tapes, and can be reduced to nine states. Ikeno's machine requires six symbols and ten states, giving a state-symbol product of 60.

A word should perhaps be said in regard to the two extra tapes in the case of Moore's machine and the four extra symbols used in Ikeno's machine. In either case, the universal machine is just slightly different from the target machines it imitates. It would be more "authentic" for a universal machine to be immediately within the class of all target machines it imitates. It also seems that it is as direct to construct such an "authentic" machine as otherwise. For these reasons, we include here the construction of a universal W-machine as an example.

Let U denote the universal W-machine to be constructed. Let the squares on the tape of U be divided into two classes: a-squares and b-squares. If the squares are numbered beginning with one for the left-most square and proceeding to the right, then the a-squares are the odd-numbered squares and the b-squares the even-numbered squares. The b-squares are there to serve as markers. The description of the target machine together with the data occupies only the a-squares. In order to clarify coding, the contents of a-squares are underlined. Thus $\underline{1}\ \underline{0}$ would mean a mark and a blank in adjacent a-squares, the content of the inbetween b-square has been left unspecified.

The W-machine instructions are coded as follows:

Instruction	*Code*
* (Stop)	$\underline{1}$
+	$\underline{1}\ \underline{1}$
−	$\underline{1}\ \underline{1}\ \underline{1}$
m	$\underline{1}\ \underline{1}\ \underline{1}\ \underline{1}$
c	$\underline{1}\ \underline{1}\ \underline{1}\ \underline{1}\ \underline{1}$
$t(n)$	$\underline{1}\ \underline{1}\ \underline{1}\ \underline{1}\ \underline{1}\ \underbrace{\underline{1}\ \underline{1}\ \cdots\ \underline{1}}_{n \text{ ones}}$

The program and data of the target W-machine that the universal W-machine U is to imitate occupy only the a-squares on the tape of U. The instructions are coded in sequence, with a single blank a-square separating adjacent instructions. The data go directly into a-squares without modification. There is a single blank a-square between the last instruction and the data.

The first two a-squares are blank and all a-squares to the right of the data are blank. The b-squares are all marked except for (a) the first b-square, (b) the b-square immediately to the right of the data square under scan and (c) all b-squares to the right of square x, where x is the a-square to the right of the last data a-square.

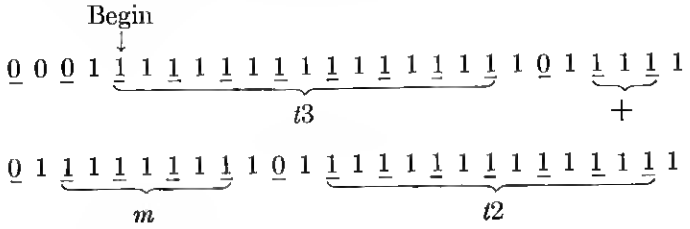
The coding scheme will be made clear by an example. Suppose the program of the target W-machine is

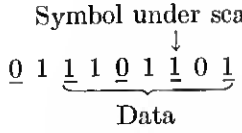
1. $t3$
2. $+$
3. m
4. $t2$,

where the initial address is address 1 and the data are

$$\overset{\downarrow}{1\ 0\ 1\ 1}$$

where the third symbol is the initial symbol under scan. In the coded form, the tape of U would have contents





The program for the universal W-machine U is divided into a main program P and a number of subprograms. The various subprograms are designated by symbolic addresses as follows:

RT	One square to the right.
LT	One square to the left.
MK	Mark square under scan.
ER	Erase square under scan.
TR	Transfer if data square under scan is marked. If transfer is effective, go to the beginning of tape and hunt to the right until the correct instruction has been found. Otherwise, go to the next instruction.
RTZ	Right to zero.
LTZ	Left to zero.
RDZ	Right to double zero.
LDZ	Left to double zero.

The program for U begins with the main program P. It first examines the instruction to be carried out. If the instruction should be $+$, $-$, m or e, the program enters subprograms RT, LT, MK or ER respectively. If the instruction should be $t(n)$, the program enters subprogram TR.

Let us begin with the basic subroutines RTZ, LTZ, RDZ and LDZ:

RTZ	1. $+2, t1$.
LTZ	1. $-2, t1$.
RDZ	1. $+2, t1, +2, t1$.
LDZ	1. $-2, t1, -2, t1$.

Next the subprograms TR, RT, LT, MK and ER:

TR	1. $+, e$, RTZ, $-, t2, -, t$(LT3), 2. LDZ, $+4$, 3. $e, +$, RTZ, $m, +, t4, +$, LTZ, $-$, RDZ, m, t(P), 4. $+, e$, LTZ, $-$, RDZ, m, RTZ, $+2, e$, LDZ, $m, t3$.
LT	1. $+, e$, RTZ, $m, -2$, 2. e, 3. LTZ, $m, +, t$(P).
RT	1. $+, e$, RTZ, $m, +2, m, t$(LT2).
MK	1. $+, e$, RTZ, $-, m, t$(ER2).
ER	1. $+, e$, RTZ, $-, e$, 2. $-, t$(LT3).

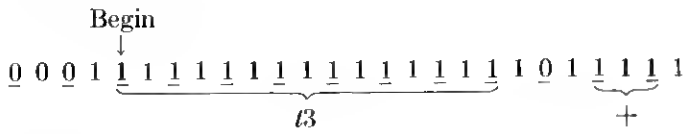
Finally, the main program P:

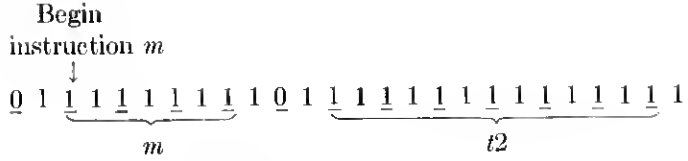
P	1. $+2, t2$, *, 2. $+2, t3$, RT, 3. $+2, t4$, LT, 4. $+2, t5$, MK, 5. $+2, t$(TR), ER.

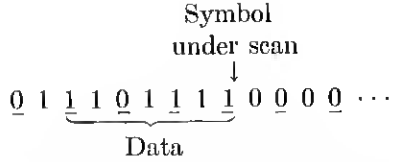
After the following sequence of instructions of the target machine has been executed:

$$t3,\ m,\ t2,\ +,$$

the tape contents read:







We will call a target W-machine *admissible* if its read-write head never goes to the left of the leftmost square on tape. Machine U then imitates all admissible target machines and is itself admissible.

It may be interesting to note that the coding for machine U does not make an intrinsic distinction between program and data. The burden of distinguishing which is program and which is data is therefore on the coder.

Using the conversion procedure discussed in the proof of Theorem 1, there is a T-machine completely equivalent to the W-machine U with about 74 internal states.† The program for U itself requires some 125 base instructions. As things go, it is not impossible for someone to improve our result to a 50 base instruction universal W-machine or a 25-state universal T-machine or perhaps even better. The answer to the problem of finding a universal machine with the smallest state-symbol product posed by Shannon[6] seems to be quite remote, even for two-symbol machines.

†Some of the ideas that resulted in this construction were due to D. Younger, who indicated a possible reduction to a machine of about 56 states.

V. FINITE AUTOMATA

There is a subfamily of T-machines that are abstract models of a class of switching circuits called sequential circuits. The dominant trait of these machines is a strictly limited memory, so that they are called *finite automata*. (These machines are also known as sequential machines.) Because of their limited memory, rather simple tasks lie beyond the reach of finite automata. For instance, there is no finite automaton that, having the null input and ejecting symbols one at a time, will give us the successive digits of π or, for that matter, any number that is not rational. On the other hand, many decision problems become finite problems for finite automata; in fact, in some cases efficient algorithms have been found.

A two-symbol finite automaton consists of

i. A finite number of internal states $q_0, q_1, \cdots, q_n$.

ii. An alphabet of two symbols: $s_0 = 0$, $s_1 = 1$.

iii. A map M whose domain and range are both subsets of the set of state-symbol pairs. If M is defined for a state-symbol pair (q_i, s_j), then $M(q_i, s_j)$ is another pair (q_k, s_r). The symbol s_j is called an *input symbol*. The symbol s_r is called an *output symbol*, and is completely determined by q_i; that is, s_r is independent of the input symbol s_j.

iv. An initial state q_0, which can reach every state q_i, $0 < i \leqq n$, via some suitable input sequence of symbols.

In the definition of a finite automaton given above, we included those automata in which the map M may be undefined for some state-symbol pairs (q_i, s_j). We will call such automata *partial automata*. Partial automata in the past have been treated somewhat differently from complete automata. By considering certain input sequences called acceptable sequences, we will be able to treat partial and complete automata on a uniform basis.

5.1 *Finite Automata and W*-Machines.*

In the beginning of this section we mentioned that finite automata can be regarded as a subfamily of T-machines, and hence as a subfamily of W-machines. Let us call a W-machine a W*-machine if the base program of the W-machine does not contain the instruction "$-$"; that is, if the read-write head of the W-machine never moves to the left. We will see that, by suitable interpretation of inputs and outputs, every finite automaton is completely equivalent to some W*-machine and, furthermore, that every W*-machine differs from some finite automaton by at most a unit of delay in the output.

Let S be a finite automaton; S may then be considered as a T-machine in the following sense: An input sequence of symbols to S corresponds to having this sequence of symbols on the tape of the T-machine, beginning with the initial input symbol on the leftmost square of the tape. In operation, the T-machine begins by scanning the initial square, writes the output symbol on the square being scanned, moves one square to the right and goes into its next state. At any moment, therefore, the previous output is contained in the square just to the left of the read-write head, and the present input is contained in the square directly under the read-write head. In this way the read-write head of the T-machine never moves to the left. It follows from Theorem 1, therefore, that there is a W-machine whose program consists of no base instruction of the form "$-$" and is such that this W-machine and the T-machine are completely equivalent.

Conversely, suppose a W*-machine is given. By Theorem 1, there is a T-machine completely equivalent to this W*-machine such that its read-write head never travels to the left. Such a T-machine may not be in the form of a finite automaton since its output symbol may be a function of both the input symbol and the current state of the machine. We wish to show therefore that such a T-machine differs from a finite automaton by at most a unit of delay in the output.

Consider a T-machine whose read-write head never travels to the left. It then consists of states of the following kind:

$$q_i;\quad a_i\,,\ +,\ q_j;\quad b_i\,,\ +,\ q_k\,,$$

where a_i and b_i are either e or m. In the particular case $a_i = b_i$ for some i, the output becomes in no way dependent upon the input. We will therefore consider only those states q_i for which $a_i \neq b_i$.

Let us now form a new T-machine by splitting each such state q_i of the original T-machine into two states, q_{i0} and q_{i1}, such that we have for the new machine,

$$q_{i0};\quad a_i\,,\ +,\ q_{j0};\quad a_i\,,\ +,\ q_{j1}$$

and

$$q_{i1};\quad b_i\,,\ +,\ q_{k0};\quad b_i\,,\ +,\ q_{k1}\,,$$

and, if q_0 should be the initial state of the original T-machine, add a new state $q_0{}^*$ as the initial state of the new machine:

$$q_0{}^*;\quad e,\ +,\ q_{00};\quad e,\ +,\ q_{01}\,.$$

In operation, the new machine imitates the original machine faithfully,

except that the output of the new machine is delayed by a unit of time; that is, the present output of the new machine is the previous output of the original machine. We have therefore

Theorem 2: Every finite automaton with s states is completely equivalent to a W*-machine with not more than $10s + 1$ base instructions. Every W*-machine with b base instructions differs from a finite automaton of not more than $2b + 1$ states by at most one unit of delay in the output.

An Example. Consider the following W*-machine:

1. $t6$
2. $+$
3. m
4. $t8$
5. e
6. $+$
7. $t2$
8. $+$
9. $+$
10. $t5$.

This W*-machine is completely equivalent to a five-state T-machine with initial state q_1:

State	Symbol	
	0	1
$*q_1$	e, $+$, q_3	m, $+$, q_7
q_3	m, $+$, q_9	m, $+$, q_9
q_7	e, $+$, q_9	m, $+$, q_3
q_9	e, $+$, q_{10}	m, $+$, q_{10}
q_{10}	e, $+$, stop	e, $+$, q_7

The T-machine is not in the form of a finite automaton, since its output symbols depend on both the state and the input symbol. Let us therefore split each state whose output symbol is different for different input symbols into two states and, in addition, define a new initial state q_0*. The machine then becomes:

State	Symbol	
	0	1
$*q_0*$	e, $+$, $q_{1,0}$	e, $+$, $q_{1,1}$
$q_{1,0}$	e, $+$, q_3	e, $+$, q_3
$q_{1,1}$	m, $+$, $q_{7,0}$	m, $+$, $q_{7,1}$
q_3	m, $+$, $q_{9,0}$	m, $+$, $q_{9,1}$
$q_{7,0}$	e, $+$, $q_{9,0}$	e, $+$, $q_{9,1}$
$q_{7,1}$	m, $+$, q_3	m, $+$, q_3
$q_{9,0}$	e, $+$, q_{10}	e, $+$, q_{10}
$q_{9,1}$	m, $+$, q_{10}	m, $+$, q_{10}
q_{10}	e, $+$, stop	e, $+$, $q_{7,1}$

This machine is identical with the original W*-machine except that its output symbols are delayed by a unit of time and its initial output symbol is always a zero. For the same input, the sequence of the tape contents of the two machines are therefore not exactly the same; the tape content of the new machine to the left of the read-write head is the tape content of the original machine to the left of the read-write head translated one square to the right. The tape contents of the two machines to the right of the read-write head are, of course, the same.

Since the output of a finite automaton depends only on its state, and since the symbol + is redundant, the state-symbol table of a finite automaton can be simplified. For instance, the nine-state machine given in the example can be given by:

State	Symbol		Output
	0	1	
$*q_0*$	$q_{1,0}$	$q_{1,1}$	0
$q_{1,0}$	q_3	q_3	0
$q_{1,1}$	$q_{7,0}$	$q_{7,1}$	1
q_3	$q_{9,0}$	$q_{9,1}$	1
$q_{7,0}$	$q_{9,0}$	$q_{9,1}$	0
$q_{7,1}$	q_3	q_3	1
$q_{9,0}$	q_{10}	q_{10}	0
$q_{9,1}$	q_{10}	q_{10}	1
q_{10}	stop	$q_{7,1}$	0

For complete automata, except for including the initial state in our model, this description is the same as that given by Moore.[7] In the same way, the description of the five-state T-machine in the example which is completely equivalent to the original W*-machine can also be simplified. We may write

State	Symbol		Outputs	
	0	1		
$*q_1$	q_3	q_7	0	1
q_3	q_9	q_9	1	1
q_7	q_9	q_3	0	1
q_9	q_{10}	q_{10}	0	1
q_{10}	stop	q_7	0	0

where to each state may be associated two output symbols, one for each input symbol. This description is essentially the model of sequential machines used by Huffman[8] and Mealy.[9] It is quite clear from the foregoing that there is a close relationship between these two models, and that one may go freely from one to the other.†

† Another way of relating models of finite automata is discussed by Cadden.[10]

5.2 *Finite Automata with a Minimum Number of States*

A problem of interest to switching circuit designers is finding finite automata having a smallest number of states. In relay circuit design, for example, the number of relays needed is usually a monotone function of the number of states the circuit has. For such circuits, therefore, the number of states becomes in a way a measure of cost.

Let A be a partial finite automaton. A finite sequence is said to be an *acceptable sequence* for A if there is an output sequence and a terminating state when this sequence is presented as the input sequence to A, with A beginning in its initial state. We will call the set of all acceptable sequences for A the *acceptable set* for A and denote this set by $R(A)$. Now let A and B be two partial finite automata and let the intersection $R_{AB} = R(A) \wedge R(B)$ be called the *common acceptable set* for A and B. Then A and B are said to be *completely equivalent* with respect to R_{AB} if, for all input sequences belonging to R_{AB}, A and B give identical output sequences. If R is a subset of R_{AB}, then equivalence of A and B with respect to R is defined similarly. It is clear that this definition of complete equivalence is the same as that given before for T- and W-machines, except the input sequences are now restricted to just the acceptable sequences.

As an example consider A and B defined as

A:

State	Symbol		Output
	0	1	
$*a_0$	a_1		1
a_1		a_1	0

B:

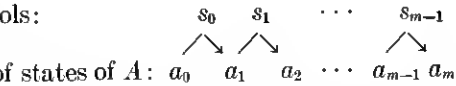
State	Symbol		Output
	0	1	
$*b_0$		b_1	1
b_1		b_2	0
b_2	b_1		0

The acceptable set $R(A)$ for A is the set of all finite sequences {0, 01, 011, 0111, · · · } and the acceptable set $R(B)$ for B is the set of finite sequences {1, 11, 110, 1101, 11010, 110101, · · · }. There is no sequence that is acceptable to both A and B. The common acceptable set R_{AB} is therefore empty.

Theorem 3: Let A and B be two partial finite automata with a and b states respectively, where $a, b > 1$. Let R_{AB} be the common acceptable set for A and B and let $R_{AB}(l)$ be the subset of R_{AB} such that every sequence in $R_{AB}(l)$ is of length $\leqq l$. Then A and B are completely equivalent with respect to R_{AB} if and only if they are completely equivalent with respect to $R_{AB}(l)$ for $l = ab - 2$.

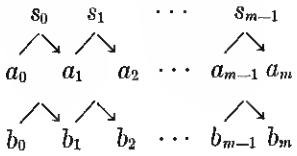
Before going through the proof, it would be helpful to discuss some

notations that will be used later. If A is a finite automaton, its initial state will be denoted by a_0. If q is any state of A, we will denote the output symbol associated with the state q by $\omega(q)$. Moreover, let $s = (s_0, s_1, \cdots, s_{m-1})$ be an acceptable input sequence for A. Then we will at times let a_i stand for the state reached by A after receiving the ith symbol of s. It will be convenient here to speak of the motion diagram for A:



Proof: The theorem is clear in one direction. In the other direction, let A and B be completely equivalent with respect to the set $R_{AB}(ab - 2)$; that is, A and B will give identical output sequences to every commonly acceptable sequence of length not greater than $ab - 2$.

Let us now suppose that there is a common acceptable sequence $s = (s_0, s_1, \cdots, s_{m-1})$ of minimum length m where $m > ab - 2$ such that in the motion diagram for A and B we have

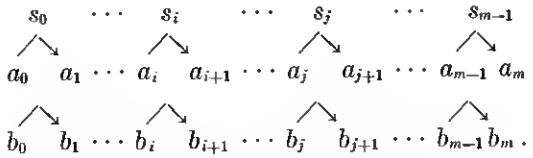


where

$$\omega(a_i) = \omega(b_i) \quad \text{for} \quad i = 0, 1, \cdots, m-1 \quad \text{but} \quad \omega(a_m) \neq \omega(b_m).$$

There are now two cases to consider. The case $m > ab - 1$ is simpler and will be left to the reader.

Let us therefore assume $m = ab - 1$. In the motion diagram above, we have then exactly ab pairs of states: (a_0, b_0), (a_1, b_1), $\cdots$, (a_m, b_m). First, suppose that these ab pairs are not distinct; that is, suppose $(a_i, b_i) = (a_j, b_j)$ for some $0 \leqq i < j \leqq m$. The motion diagram then becomes



Consider the common acceptable sequence $s^* = (s_0, \cdots, s_{i-1}, s_j,$

$\cdots$, s_{m-1}), which is of length l, where $l < m = ab - 1$. Since $\omega(a_m) \neq \omega(b_m)$, A and B would give different output sequences to the input sequence s^*, contradicting our hypothesis that A and B are completely equivalent with respect to R_{AB} $(ab - 2)$. We must therefore assume that the ab pairs of states $(a_0, b_0), \cdots, (a_m, b_m)$ are distinct, and thus include every possible pair of states of A and B.

Now let a_m' and b_m' be states of A and B respectively such that $a_m' \neq a_m$ and $b_m' \neq b_m$. Then we assert $\omega(a_m) \neq \omega(a_m')$ and $\omega(b_m) \neq \omega(b_m')$. For, if $\omega(a_m) = \omega(a_m')$, then $\omega(a_m') \neq \omega(b_m)$. This is impossible, however, since the pair (a_m', b_m) is one of the ab distinct pairs of states. The same argument shows $\omega(b_m) \neq \omega(b_m')$. We have now then the inequality $\omega(a_m') \neq \omega(b_m')$. But again this is impossible. This concludes the proof.

Although we cannot say that the bound $ab - 2$ is the best for all pairs (a, b), we will show that $ab - 2$ is very close to the best we can hope for. To do this we will now exhibit a pair of families of finite automata.

Consider first a family of finite antomata $\{A_m\}$, $m \geqq 1$, as follows:

State	Symbol		Output
	0	1	
$*a_0$	a_1		0
a_1	a_2		0
$\vdots$	$\vdots$		$\vdots$
a_{m-1}	a_m		0
a_m		a_0	0

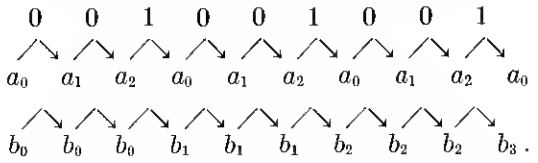
Next, define a family of finite automata $\{B_n\}$, $n \geqq 1$, as follows:

State	Symbol		Output
	0	1	
$*b_0$	b_0	b_1	0
b_1	b_1	b_2	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
b_{n-1}	b_{n-1}	b_n	0
b_n	b_n		1

For any pair of automata (A_m, B_n), one from each family, the set $R_{A_m B_n}$ of all commonly acceptable sequences consists of all sequences each of which must be of the form

$$\underbrace{0\,0\,\cdots\,0}_{m\ 0\text{'s}}\ 1\ \underbrace{0\,0\,\cdots\,0}_{m\ 0\text{'s}}\ 1\ \cdots\ \underbrace{0\,0\,\cdots\,0}_{m\ 0\text{'s}}\ 1,$$

since these are the only sequences acceptable to A_m. For these two finite automata (A_m, B_n), the minimum length of any input sequence in $R_{A_m B_n}$ that would cause A_m and B_n to give different output sequences would be $ab - \min(a, b)$, where in this case $a = m + 1$ and $b = n + 1$. For instance, the motion diagram for the pair (A_2, B_3) would be



Since b_3 is the only state of B_3 that gives an output symbol of 1, we see that the input sequence (0 0 1 0 0 1 0 0 1) is the first such sequence that causes A_2 and B_3 to give different output symbols.

In general, by the same construction, we find that given two finite automata, one from each of these families, no input sequence of length less than $ab - \min(a, b)$ would enable us to tell them apart. We therefore have

Theorem 4: Theorem 3 would not hold if l were made less than $ab - \min(a, b)$.

In particular, we note that Theorem 3 implies Theorem 4 for the case $\min(a, b) = 2$. For the cases $\min(a, b) > 2$, there may be some slight improvement† possible for Theorem 3.

Actually, Theorem 3 is interesting for another reason. It is essentially a theorem showing the existence of a decision procedure for finding finite automata with a minimum number of states. Historically, the problem of finding finite automata with a minimum number of states was studied and solved in a rather special way. Thus, both Moore[7] and Huffman[8] gave ingenious procedures for state minimization of *complete* finite automata. It was not uncommon for people to assume that these procedures also worked for partial automata before the introduction of several interesting counter-examples by Giusburg.[12] As we see from Theorem 3, much of the earlier confusion was probably due to a disregard of the idea of acceptable sequences.

VI. FINITE AUTOMATA DEFINED BY INPUT SEQUENCES

Up to now we have shown that finite automata can be described in two different ways. In the definition given in the previous section, a finite

† In the paper by Rabin and Scott,[11] a theorem similar to Theorem 3 was obtained for the family of complete automata. In view of the fact that they were dealing exclusively with complete automata, their theorem could be considerably improved.

automaton is characterized essentially by its state-symbol table. On the other hand, one may characterize a finite automaton by giving its W*-machine program. The latter characterization illustrates the close parallel between computer programming and logical design. In this section, following the earlier work of Kleene,[3] we will consider a third characterization of partial finite automata. This characterization leads to a very interesting algebraic-like structure for finite automata. Our purpose here is to connect this characterization with the others. Much of the work along the approach of Kleene had been pursued and simplified by Myhill[13] and Rabin and Scott.[11] The interested reader may refer to these papers and other unpublished work by Myhill.

Let A be a finite automaton. A finite input sequence to A is said to be a *signal sequence* for A if this input sequence causes A to terminate in a state whose output is the symbol 1. The set of all signal sequences for a finite automaton A is called the *signal set* for A, and is denoted by $\Gamma(A)$.

Given a finite automaton A, the signal set $\Gamma(A)$ is uniquely defined. On the other hand, if signal sets are to represent finite automata, it would be most desirable that two "different" automata have different signal sets. Let us consider automata A and B given by

A:

State	Symbol		Output
	0	1	
$*a_0$	a_1		1
a_1		a_1	0

B:

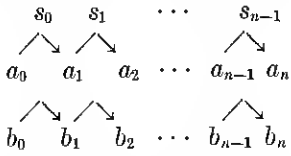
State	Symbol		Output
	0	1	
$*b_0$		b_2	1
b_1		b_2	0
b_2	b_1		0

If nothing is said about input sequences, one may say that A and B are different, since every input sequence acceptable to A is unacceptable to B and vice versa, although A and B both have the empty set as their signal set. In order to have a clear-cut correspondence between signal sets and finite automata, we must therefore restrict ourselves to acceptable sequences.

Theorem 5: Let A and B be two finite automata and R_{AB} the common acceptable set for A and B. Then A and B are completely equivalent with respect to R_{AB} if and only if A and B have the same signal set.

Proof: From the definition of signal set, it is clear that, if A and B are completely equivalent with respect to R_{AB}, then $\Gamma(A) = \Gamma(B)$. Now suppose A and B have the same signal set but are not completely equivalent with respect to R_{AB}. Then there is some input sequence

$s_0, s_1, \cdots, s_{n-1}$ in R_{AB} giving the motion diagram



such that $\omega(a_n) \neq \omega(b_n)$; that is, the output symbols associated with states a_n and b_n are different. Since we are considering only two-symbol automata, it is clear that the input sequence $s_0, s_1, \cdots, s_{n-1}$ cannot be a signal sequence for both A and B. The proof now follows from this contradiction.

We see from this that signal sets indeed represent finite automata. In many ways this is a rather natural characterization. For example, consider a sequential lock on a vault. The vault can be opened only if a given sequence s of symbols is applied to the lock. Any other sequence of input symbols may cause the lock to go into an alarm state. In this case, we may consider the lock as a finite automaton defined by the one-element signal set $\{s\}$.

There are other situations, however, where it seems simpler to describe a finite automaton by its W*-machine program or its state-symbol table. It is therefore not clear in general how a finite automaton is best characterized; as far as we can tell, a great deal depends on personal taste. The next best thing one can do, therefore, is to find ways to go from one form of characterization to another.

We will begin by redefining several operations on sets of finite sequences due to Kleene. Let X and Y be two sets of finite sequences; $X \vee Y$ is then the set *union* of X with Y. By XY, called the *string product* of X with Y, we mean the set of all concatenated finite sequences of the form xy with $x \in X$ $y \in Y$. Finally, by the *closure* of the set X, denoted by X^*, is meant the set

$$X^* = \emptyset \vee X \vee XX \vee XXX \vee \cdots,$$

where $\emptyset$ is the empty set.

To illustrate the use of these operations, let us consider the following automaton A:

State	Symbol		Output
	0	1	
$*a_0$	a_1	a_0	0
a_1		a_0	1

The signal set of A is then given by

$$\Gamma(A) = 1^* 0 (1 1^* 0)^*,$$

where we have used the notations 0 and 1 to stand for the one-element sets $\{0\}$ and $\{1\}$.

In general, to find the signal set for some finite automaton A is not as straightforward as this example indicates. We will describe below one such procedure.†

Let A be a finite automaton with k states $a_0, a_1, \cdots, a_{k-1}$. Then by $P(a_i, a_j)$ we mean the set of all finite sequences such that beginning with state a_i, each of these sequences causes A to terminate in state a_j. Furthermore, let us denote by $P(a_i, a_j; a_j')$ the set of all finite sequences such that, beginning with state a_i, each of these sequences not only causes A to terminate in state a_j, but also never causes A to pass through state a_j. In other words, it is permissible for $a_i = a_j$, but in the chain of states $a_i, \cdots, a_j$, the state a_j must not appear other than at either end. Then it is clear that

Lemma 2: $P(a_i, a_j) = P(a_i, a_j; a_j')\ [P(a_j, a_j; a_j')]^*$.

More generally, let $s_0, s_1, \cdots, s_{n-1}$ be a sequence in $P(a_i, a_j)$ with the motion diagram

$$\begin{array}{ccccccc} & s_0 & s_1 & & \cdots & & s_{n-1} \\ & \swarrow & \swarrow & & & & \swarrow \\ a_i & & a_{j_1} & a_{j_2} & \cdots & a_{j_{n-1}} & a_j . \end{array}$$

We denote by $P(a_i, a_j; a_{i_1}', a_{i_2}', \cdots, a_{i_m}')$ a subset of $P(a_i, a_j)$ such that a sequence $s_0, s_1, \cdots, s_{n-1}$ is in $P(a_i, a_j; a_{i_1}', a_{i_2}', \cdots, a_{i_m}')$ if and only if the two sets of states $(a_{i_1}, a_{i_2}, \cdots, a_{i_m})$ and $(a_{j_1}, a_{j_2}, \cdots, a_{j_{n-1}})$ are disjoint. In other words, $P(a_i, a_j; a_{i_1}', a_{i_2}', \cdots, a_{i_m}')$ is the set of finite sequences such that, beginning with state a_i, each of these sequences not only causes A to terminate in state a_j but also causes A never to go through states $a_{i_1}, a_{i_2}, \cdots, a_{i_m}$. It is permissible, however, for a_i or a_j to be one of the states $a_{i_1}, a_{i_2}, \cdots, a_{i_m}$.

Lemma 3: Let A be a finite automaton with k states $a_0, a_1, \cdots, a_{k-1}$. Then, for all pairs of states a_i, a_j, and for all m, $1 \leqq m \leqq k - 1$,

$$P(a_i, a_j; a_{i_1}', a_{i_2}', \cdots, a_{i_m}') =$$

$$P(a_i, a_j; a_{i_1}', \cdots, a_{i_{m+1}}') \vee P(a_i, a_{i_{m+1}}; a_{i_1}', \cdots, a_{i_{m+1}}')$$

$$[P(a_{i_{m+1}}, a_{i_{m+1}}; a_{i_1}', \cdots, a_{i_{m+1}}')]^* P(a_{i_{m+1}}, a_j; a_{i_1}', \cdots, a_{i_{m+1}}').$$

Proof: Suppose that an input sequence belongs to the set on the left-hand side. Then this sequence causes A to either go through state $a_{i_{m+1}}$

† In an unpublished report shown to me by H. Wang, I found a similar result worked out independently by R. McNaughton and H. Yamada.

or it does not. If it does not, then it clearly belongs to $P(a_i, a_j; a'_{i_1}, \cdots, a'_{i_{m+1}})$. If it does, then it belongs to the second set on the right hand side. Conversely, suppose a sequence belongs to the set on the right hand side. Then it clearly belongs to $P(a_i, a_j; a'_{i_1}, \cdots, a'_{i_m})$, and the proof follows.

Combining the two lemmas, we get

Theorem 6: Let A be a finite automaton with k states $a_0, a_1, \cdots, a_{k-1}$. Let a_0 be the initial state of A and $a_{r_1}, a_{r_2}, \cdots, a_{r_n}$ be all those states of A whose output symbol is one. Then the signal set for A is the union

$$\Gamma(A) = \bigvee_{i=1}^{n} P(a_0, a_{r_i}; a'_{r_i}) [P(a_{r_i}, a_{r_i}; a'_{r_i})]^*,$$

which can be obtained by repeated application of Lemma 3.

As an illustration, let us consider the automaton A below:

State	Symbol		Output
	0	1	
a_0	a_0	a_1	1
a_1	a_1	a_2	1
a_2	a_2	a_1	0

By Lemma 2, we have

$$\Gamma(A) = P(a_0, a_0; a'_0) [P(a_0, a_0; a'_0)]^* \vee P(a_0, a_1; a'_1) [P(a_1, a_1; a'_1)]^*.$$

Now

$$P(a_0, a_0; a'_0) = 0,$$

$$P(a_0, a_1; a'_1) = 0^* 1,$$

$$P(a_1, a_1; a'_1) = 0 \vee 1 0^* 1.$$

Therefore,

$$\Gamma(A) = 0 0^* \vee 0^* 1 (0 \vee 1 0^* 1)^*.$$

The expressions for signal sets can get very lengthy. The problem of reducing the length of these expressions without recourse to an exhaustive search appears very difficult and intriguing.

The next problem we will consider is how to give a state-symbol characterization of signal sets. The procedure we will describe here is a modification of the abstract ideas of Rabin and Scott[11] and Myhill.[13]

Let us begin this discussion of several examples. Let A and B be the following finite automata:

A:

State	Symbol		Output
	0	1	
$*a_0$	a_1	a_0	0
a_1		a_0	1

B:

State	Symbol		Output
	0	1	
$*b_0$	b_1	b_1	0
b_1	b_0	b_0	1

with signal sets $\Gamma(A) = 1^* 0 (1 1^* 0)^*$ and $\Gamma(B) = (0 \vee 1)[(0 \vee 1)(0 \vee 1)]^*$.

Example 1. Suppose we wish to construct an automaton C such that $\Gamma(C) = \Gamma(A) \vee \Gamma(B)$. We begin by defining a set of new states (a_0, b_0), (a_0, b_1), (a_1, b_0), (a_1, b_1), some of which may turn out to be superfluous. The state (a_0, b_0) is defined to be the initial state of C. Beginning with the state (a_0, b_0), we can construct a part of C:

State	Symbol		Output
	1	0	
(a_0, b_0)	(a_1, b_1)	(a_0, b_1)	$\omega(a_0) \vee \omega(b_0) = 0$

where, if we let M_C denote the function taking state-symbol pairs to states for the automaton C and ω be the function taking states to output symbols, then

$$M_C[(a_0, b_0), 0] = (M_A(a_0, 0), M_B(b_0, 0)),$$

$$M_C[(a_0, b_0), 1] = (M_A(a_0, 1), M_B(b_0, 1))$$

and

$$\omega[(a_0, b_0)] = \omega(a_0) \vee \omega(b_0).$$

In this process, we reached two new states (a_1, b_1) and (a_0, b_1). Continuing the process, we eventually get for C the state-symbol table

State	Symbol		Output
	0	1	
$*(a_0 , b_0)$	(a_1 , b_1)	(a_0 , b_1)	0
(a_0 , b_1)	(a_1 , b_0)	(a_0 , b_0)	1
(a_1 , b_1)	b_0	(a_0 , b_0)	1
(a_1 , b_0)	b_1	(a_0 , b_1)	1
b_0	b_1	b_1	0
b_1	b_0	b_0	1

Let us suppose that an input sequence $s_0 , s_1 , \cdots , s_{n-1}$ belongs to $\Gamma(A)$ and gives the motion diagram

$$\begin{array}{ccccccc} & s_0 & s_1 & & \cdots & & s_{n-1} \\ & \swarrow & \searrow\swarrow & & & & \searrow \\ a_0 & & a_{i_1} & a_{i_2} & \cdots & a_{i_{n-1}} & a_{i_n}. \end{array}$$

Then the same sequence would give rise to a chain of states of C such that the terminal state of this chain must be (a_{i_n} , b_j) for some state b_j of B. Since

$$\omega(a_{i_n} , b_j) = \omega(a_{i_n}) \vee \omega(b_j) = 1,$$

it follows that this input sequence belongs to $\Gamma(C)$ and $\Gamma(A) \subset \Gamma(C)$. In the same way, we may show that $\Gamma(B) \subset \Gamma(C)$.

Conversely, if a sequence in $\Gamma(C)$ gives a chain of states of C: (a_0 , b_0), (a_{i_1} , b_{i_1}), $\cdots$, (a_{i_n} , b_{i_n}), then either $\omega(a_{i_n}) = 1$ or $\omega(b_{i_n}) = 1$. Therefore, this input sequence is either in $\Gamma(A)$ or $\Gamma(B)$, and thus $\Gamma(A) \vee \Gamma(B) = \Gamma(C)$.

Example 2. We wish to construct a finite automaton C such that $\Gamma(C) = \Gamma(A)\Gamma(B)$. We begin with the initial state of A as the initial state for C. Now, whenever a state of A is reached whose output symbol is a 1, we must then allow C the opportunity to imitate the behavior of B. In such cases, therefore, new states may be created. Thus, a part of the state-symbol table for C would be

State	Symbol		Output
	0	1	
$*a_0$	a_1	a_0	0
a_1	b_1	(a_0 , b_1)	1

The state (a_0 , b_1) is defined by $(M_A(a_1 , 1), M_B(b_0 , 1))$. In this way, the new state allows C to imitate immediately the behavior of state b_0 of B. Also, if either $\omega(a_0) = 1$ or $\omega(b_1) = 1$, then $\omega(a_0 , b_1) = 1$. We may therefore continue this process to get for C the state-symbol table:

State	Symbol		Output
	0	1	
$*a_0$	a_1	a_0	0
a_1	b_1	(a_0, b_1)	1
b_1	b_0	b_0	0
b_0	b_1	b_1	1
(a_0, b_1)	(a_1, b_0)	(a_0, b_0)	1
(a_1, b_0)	b_1	(a_0, b_1)	1
(a_0, b_0)	(a_1, b_1)	(a_0, b_1)	0
(a_1, b_1)	(b_1, b_0)	(a_0, b_0, b_1)	1
(b_1, b_0)	(b_0, b_1)	(b_0, b_1)	1
(a_0, b_0, b_1)	(a_1, b_0, b_1)	(a_0, b_0, b_1)	1
(a_1, b_0, b_1)	(b_1, b_0)	(a_0, b_0, b_1)	1

where we see that $(a_1, b_0, b_1) = (a_1, b_1)$.

The process can be formulated as follows: If $(a_i, \cdots, a_j, b_k, \cdots, b_m)$ is a new state, then

$$M_C[(a_i, \cdots, a_j, b_k, \cdots, b_m), x]$$
$$= (M_C(a_i, x), \cdots, M_C(a_j, x), M_B(b_k, x), \cdots, M_B(b_m, x))$$

where x is either 0 or 1, and $\omega[(a_i, \cdots, a_j, b_k, \cdots, b_m)] = \omega(a_i) \vee \cdots \vee \omega(b_k) \vee \cdots \vee \omega(b_m)$. Also, if a_i is any state such that $\omega(a_i) = 1$, then $M_C(a_i, x) = (M_A(a_i, x), M_B(b_0, x))$. For all other a_j and for all b_k we have

$$M_C(a_j, x) = M_A(a_j, x),$$
$$M_C(b_k, x) = M_B(b_k, x),$$

where x is again either 0 or 1.

Example 3. We wish to construct an automaton C such that $\Gamma(C) = [\Gamma(A)]^*$. The idea here is that whenever a state of A is reached whose output symbol is a 1, we must allow C the opportunity to begin again at state a_0 of A. Furthermore, since the empty sequence is a member of $\Gamma(C)$, it is necessary to define for C a new initial state C_0 whose output symbol is 1. Following this line of thought, we see that the state-symbol table for C is

State	Symbol		Output
	0	1	
$*c_0$	a_0	a_0	1
a_0	a_1	a_0	0
a_1	a_1	a_0	1

In general, the process is formulated as follows. If $(a_i, \cdots, a_j)$ is a new state, then

$$M_C[(a_i, \cdots, a_j), x] = (M_C(a_i, x), \cdots, M_C(a_j, x)),$$

where x is either 0 or 1 and

$$\omega[(a_i, \cdots, a_j)] = \omega(a_i) \vee \cdots \vee \omega(a_j).$$

If a_i is any state of A whose output symbol is 1,

$$M_C(a_i, x) = (M_A(a_i, x), M_A(a_0, x)), \qquad x = 0 \text{ or } 1.$$

For all other states a_j of A,

$$M_C(a_j, x) = M_A(a_j, x), \qquad x = 0 \text{ or } 1.$$

The ideas of conversion from signal sets to state-symbol table for a finite automaton are all contained in these examples. Since to state a theorem means a repetition of what we outlined in the examples, we will content ourselves with the following form of Kleene's result.[3]

Remark. Let $\Gamma(A)$ be a set of finite sequences built up from the operations union, string product and closure operating on a finite set of finite sequences. That is, $\Gamma(A)$ is given by a finite expression involving the operations union, string product and closure. Then, following the procedures outlined in Examples 1, 2 and 3, a finite automaton can be constructed having $\Gamma(A)$ as its signal set.

This remark, together with Theorem 6, thus provides the two-way linkage between finite automata and signal sets.

VII. CONCLUDING REMARKS

We have discussed three approaches to a theory of automata and finite automata: the state-symbol table model, the W-machine program model and the signal-set model. Of these, we are most intrigued by the programming model. This approach not only resembles strongly computer programming, but it also offers possibilities of symbol operation and other combinatorial programs, all based on a very simple and elegant program structure. (One other model not studied here is a system proposed by Post.) It is quite possible a combination of these systems may offer deeper insight into the global structure of programming and automata which is lacking at present.

VIII. ACKNOWLEDGMENT

The writer is indebted to E. F. Moore and T. H. Crowley of Bell Telephone Laboratories for their suggestions which led to an improved version of Theorem 4.

REFERENCES

1. Turing, A. M., On Computable Numbers, with an Application to the Entscheidungs Problem, Proc. London Math. Soc., **24,** 1936, p. 230.
2. Wang, H., A Variant to Turing's Theory of Computing Machines, Jour. A.C.M., **4,** 1957, p. 63.
3. Kleene, S. C., Representation of Events in Nerve Nets and Finite Automata, *Automata Studies*, Annals of Math. Studies, No. 34, Princeton Univ. Press, Princeton, N. J., 1956, p. 3.
4. Moore, E. F., A Simplified Universal Turing Machine, Proc. A.C.M. (Sept. 8, 1952), 1953.
5. Ikeno, N., An Example of Universal Turing Machine, (in Japanese), Proc. Inst. Elec. Comm. of Japan, July 1958.
6. Shannon, C. E., A Two-State Universal Turing Machine, *Automata Studies* Annals of Math. Studies, No. 34, Princeton Univ. Press., Princeton, N. J., 1956, p. 157.
7. Moore, E. F., Gedanken Experiments on Sequential Machines, *Automata Studies*, Annals of Math. Studies, No. 34, Princeton Univ. Press, Princeton, N. J., 1956, p. 129.
8. Huffman, D. A., The Synthesis of Sequential Switching Circuits, J. Frank. Inst., **257,** 1954, pp. 161; 275.
9. Mealy, G. H., A Method for Synthesizing Sequential Circuits, B.S.T.J., **34,** 1955, p. 1045.
10. Cadden, W. J., Equivalent Sequential Circuits, I.R.E. Trans., **CT-6,** 1959, p. 30.
11. Rabin, M. O. and Scott, D., Finite Automata and Their Decision Problems, I.B.M. J. Res. & Dev., **3,** 1959, p. 114.
12. Ginsburg, S., On the Reduction of Superfluous States in a Sequential Machine, Jour. A.C.M., **6,** 1959, p. 259.
13. Myhill, J., Finite Automata and Representation of Events, W.A.D.C. Report, 1957.